IS 6733 Deep Learning on Cloud Platforms

Lecture 2b Python Tutorial - Pandas

Dr. Yuanxiong Guo
Department of Information Systems and Cyber Security



Python Checklist in Machine Learning

- Essential libraries and tools in data science
 - Jupyter Notebook/Colab
 - NumPy
 - Pandas
 - Matplotlib
 - Scikit-Learn
 - Keras/TensorFlow

Pandas

 [pandas] is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals. — Wikipedia

This tool is essentially your data's home.
 Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it.

What's Pandas for?

- Say you want to explore a dataset stored in a CSV on your computer. Pandas will extract the data from that CSV into a DataFrame — a table, basically — then let you do things like:
 - Calculate statistics and answer questions about the data, like
 - What's the average, median, max, or min of each column?
 - Does column A correlate with column B?
 - What does the distribution of data in column C look like?
 - Clean the data by doing things like removing missing values and filtering rows or columns by some criteria
 - Visualize the data with help from Matplotlib. Plot bars, lines, histograms, bubbles, and more.
 - Store the cleaned, transformed data back into a CSV, other file or database

How dose Pandas fit into the data science toolkit?

- Pandas is built on top of the NumPy package, meaning a lot of the structure of NumPy is used or replicated in Pandas. Data in pandas is often used to feed statistical analysis in SciPy, plotting functions from Matplotlib, and machine learning algorithms in Scikit-learn.
- Jupyter Notebooks offer a good environment for using pandas to do data exploration and modeling, but pandas can also be used in text editors just as easily.

When should you start using pandas?

- If you do not have any experience coding in Python, then you should stay away from learning pandas until you do.
- You don't have to be at the level of the software engineer, but you should be adept at the basics, such as lists, tuples, dictionaries, functions, and iterations.
- Familiarize yourself with **NumPy** due to the similarities mentioned above.

Install and Import

- conda install pandas
- pip install pandas
- !pip install pandas (in a Jupyter notebook)
 - The! at the beginning runs cells as if they were in a terminal.
 - Common libraries such as pandas, numpy, matploblib have already been installed in Colab. Therefore, you can directly import and use them.
- import pandas as pd

Core Components of Pandas

- Two primary two components of Pandas: Series and DataFrame
- A Series is essentially a column, and a DataFrame is a multidimensional table made up of a collection of Series.
- A Series is an analog of a one-dimensional array with flexible indices.
- A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.

Series			Series			DataFrame		
	apples			oranges			apples	oranges
0	3		0	0		0	3	0
1	2	+	1	3	=	1	2	3
2	0		2	7		2	0	7
3	1		3	2		3	1	2

Creating Series from Scratch

 A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
data

Out[2]: 0     0.25
     1     0.50
     2     0.75
     3     1.00
     dtype: float64
```

As we see in the output, the Series wraps both a sequence of values and a sequence
of indices, which we can access with the values and index attributes.

Series as generalized NumPy array

While the NumPy Array has an *implicitly defined* integer index used to access the values, the Pandas Series has an *explicitly defined* index associated with the values.

```
In [7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                                                                             we can use strings as an index
                           index=['a', 'b', 'c', 'd'])
          data
Out[7]: a
              0.25
              0.50
               0.75
          C
              1.00
         dtype: float64
 In [8]: data['b']
 Out[8]: 0.5
 In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                            index=[2, 5, 3, 7])
                                                                     We can use non-contiguous or non-sequential indices.
           data
 Out[9]: 2
               0.25
                0.50
                0.75
                1.00
          dtype: float64
In [10]: data[5]
Out[10]: 0.5
                                                                                                                10
```

Series as specialized dictionary

A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and
a Series is a structure which maps typed keys to a set of typed values.

```
In [11]: population_dict = {'California': 38332521,
                                 'Texas': 26448193,
                                                                        We can also construct a Series object directly from a
                                 'New York': 19651127,
                                 'Florida': 19552860,
                                                                        Python dictionary.
                                 'Illinois': 12882135}
             population = pd.Series(population dict)
            population
Out[11]: California
                           38332521
            Florida
                           19552860
            Illinois
                           12882135
            New York
                           19651127
                           26448193
            Texas
            dtype: int64
                                                                      Typical dictionary-style item access can be performed.
In [12]:
            population['California']
Out[12]: 38332521
           population['California':'Illinois']
                                                                    Unlike a dictionary, though, the Series also supports array-
In [13]:
                                                                    style operations such as slicing.
Out[13]: California
                          38332521
           Florida
                          19552860
           Illinois
                          12882135
           dtype: int64
```

Constructing Series objects

```
>>> pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities.

For example, data can be a list or NumPy array, in which case index defaults to an integer sequence:

data can be a scalar, which is repeated to fill the specified index:

data can be a dictionary

In each case, the index can be explicitly set if a different result is preferred:

Notice that in this case, the Series is populated only with the explicitly identified keys.

DataFrame as generalized NumPy array

If a Series is an analog of a one-dimensional array with flexible indices,
 a DataFrame is an analog of a two-dimensional array with both flexible row indices
 and flexible column names.

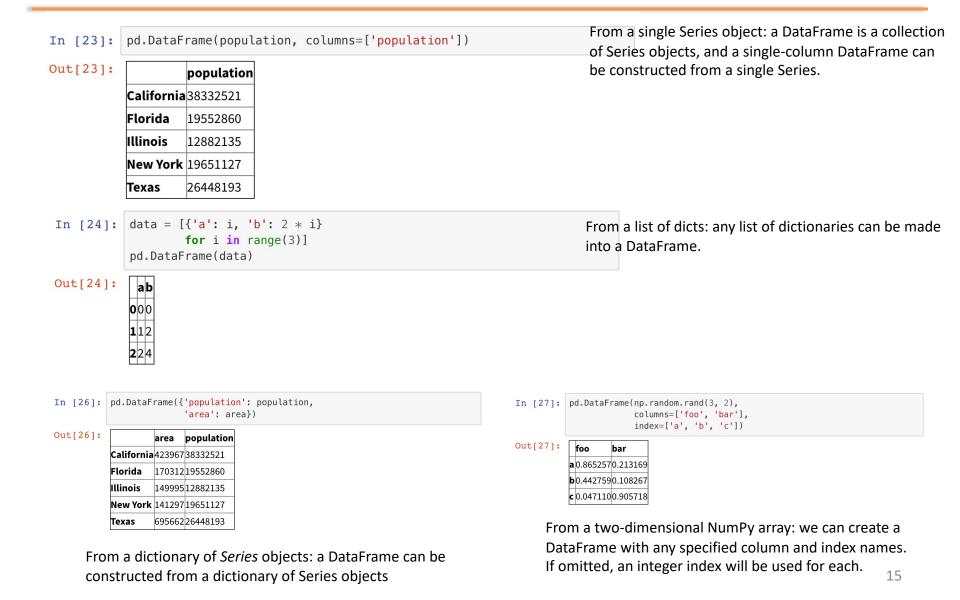
```
In [18]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297
                         'Florida': 170312, 'Illinois': 149995}
           area = pd.Series(area_dict)
           area
Out[18]: California
                          423967
           Florida
                          170312
           Illinois
                          149995
           New York
                          141297
           Texas
                          695662
           dtype: int64
In [19]: states = pd.DataFrame({'population': population,
                                   'area': area})
           states
Out[19]:
                    area population
           California 423967 38332521
           Florida
                    170312 19552860
                    149995 12882135
            Illinois
            New York | 141297 | 19651127
                    695662 26448193
            Texas
 In [20]: states.index
 Out[20]: Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtyr
  In [21]: states.columns
  Out[21]: Index(['area', 'population'], dtype='object')
```

DataFrame as specialized dictionary

 Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data.

• Potential point of confusion: in a two-dimensional NumPy array, data[0] will return the first row. For a DataFrame, data['col0'] will return the first column.

Constructing *DataFrame* objects



Data Indexing and Selection

- How to assess and modify values in Pandas Series and DataFrame objects?
- Similar to NumPy in most cases, but there are a few quirks to be aware of.
- Remember that both Series and DataFrame objects can be thought as generalized NumPy array or specialized dictionary. Therefore, their operations are similar in many ways.

Data Selection in Series

- A Series object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary.
- Series as dictionary

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
In [3]: 'a' in data
Out[3]: True
In [4]: data.keys()
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')
In [5]: list(data.items())
Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

You can extend a Series by assigning to a new index value:

```
In [6]: data['e'] = 1.25
    data

Out[6]: a     0.25
     b     0.50
     c     0.75
     d     1.00
     e     1.25
     dtype: float64
```

Data Selection in Series

Series as one-dimensional array

```
In [7]: # slicing by explicit index
                                                                               When slicing with an explicit index (i.e., data['a':'c']),
          data['a':'c']
                                                                               the final index is included in the slice.
               0.25
Out[7]: a
               0.50
               0.75
          dtype: float64
                                                                               when slicing with an implicit index (i.e., data[0:2]),
In [8]: # slicing by implicit integer index
          data[0:2]
                                                                               the final index is excluded from the slice.
Out[8]: a
               0.25
               0.50
          dtype: float64
 In [9]: # masking
           data[(data > 0.3) & (data < 0.8)]
 Out[9]: b
                0.50
                0.75
           dtype: float64
In [10]: # fancy indexing
           data[['a', 'e']]
                0.25
Out[10]: a
               1.25
           dtype: float64
```

To avoid the potential confusion in explicit and implicit indices, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes.

Data Selection in Series

- To avoid the potential confusion in explicit and implicit indices, Pandas provides some special *indexer* attributes (*loc*, *iloc*, and *ix*) that explicitly expose certain indexing schemes.
- The loc attribute allows indexing and slicing that always references the
 explicit index, while the iloc attribute allows indexing and slicing that always
 references the implicit Python-style index. A third indexing attribute, ix, is a
 hybrid of the two, and for Series objects is equivalent to standard []-based
 indexing.

Data Selection in DataFrame

- A DataFrame acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of Series structures sharing the same index.
- DataFrame as a dictionary

Out[18]: area pop

California 423967 38332521

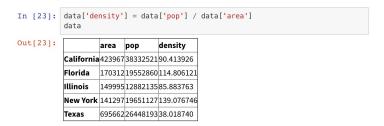
Florida 170312 19552860

Illinois 149995 12882135

New York 141297 19651127

Texas 695662 26448193

Adding a new column:



The columns of the DataFrame can be accessed via dictionarystyle indexing of the column name:

We can use attribute-style access only with column names that are strings:

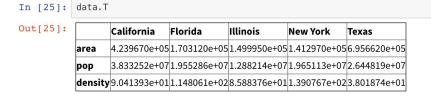
Data Selection in DataFrame

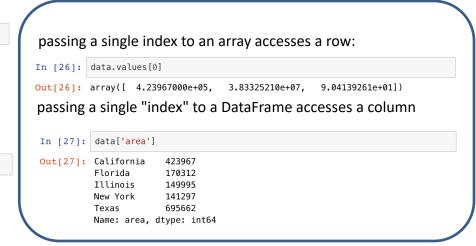
DataFrame as two-dimensional array

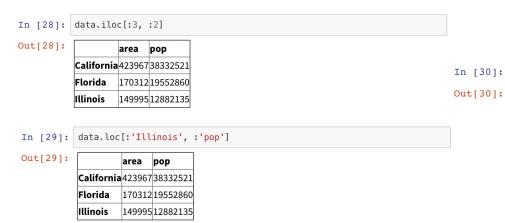
Use the values attribute to examine the raw underlying data array

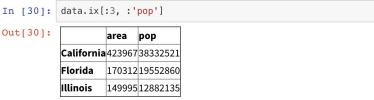
```
In [24]: data.values
Out[24]: array([[ 4.23967000e+05,
                                     3.83325210e+07,
                                                       9.04139261e+01],
                 [ 1.70312000e+05,
                                     1.95528600e+07,
                                                       1.14806121e+02],
                 [ 1.49995000e+05,
                                     1.28821350e+07,
                                                       8.58837628e+01],
                 [ 1.41297000e+05.
                                     1.96511270e+07.
                                                       1.39076746e+02].
                                     2.64481930e+07.
                 [ 6.95662000e+05.
                                                       3.80187404e+01]])
```

Can transpose the full DataFrame to swap rows and columns









How to read in data

- Reading data from CSVs
 - With CSV files all you need is a single line to load in the

data:

```
df = pd.read_csv('purchases.csv')
df
```

 Unnamed: 0
 apples
 oranges

 0
 June
 3
 0

 1
 Robert
 2
 3

 2
 Lily
 0
 7

 3
 David
 1
 2

CSVs don't have indexes like our DataFrames, so all we need to do is just designate the index_col when

reading:

```
df = pd.read_csv('purchases.csv', index_col=0)
df
```

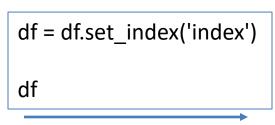
How to read in data

Reading data from JSON

```
df = pd.read_json('purchases.json')
df
```

- Reading data from a SQL database
 - If you're working with data from a SQL database you need to first establish a connection using an appropriate Python library, then pass a query to pandas.

	index	apples	oranges
0	June	3	0
1	Robert	2	3
2	Lily	0	7
3	David	1	2



	apples	oranges
index		
June	3	0
Robert	2	3
Lily	0	7
David	1	2

we could use set_index() on *any* DataFrame using *any* column at *any* time. Indexing Series and DataFrames is a very common task, and the different ways of doing it is worth remembering.

Converting back to a CSV, JSON, or SQL

 After extensive work on cleaning your data, you're now ready to save it as a file of your choice.

```
df.to_csv('new_purchases.csv')

df.to_json('new_purchases.json')

df.to_sql('new_purchases', con)
```

- When we save JSON and CSV files, all we have to input into those functions is our desired filename with the appropriate file extension.
- With SQL, we're not creating a new file but instead inserting a new table into the database using our con variable from before.

 DataFrames possess hundreds of methods and other operations that are crucial to any analysis.

 As a beginner, you should know the operations that perform simple transformations of your data and those that provide fundamental statistical analysis.

Let's load in the IMDB movies dataset to begin:

```
movies_df = pd.read_csv("IMDB-Movie-Data.csv", index_col="Title")
```

- View your data
 - The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference.

movies_df.head()
movies_df.tail(2)

	Rank	Genre	Description	Director	Actors	Year
Title						
Search Party	999	Adventure,Comedy	A pair of friends embark on a mission to reuni	Scot Armstrong	Adam Pally, T.J. Miller, Thomas Middleditch,Sh	2014
Nine Lives	1000	Comedy,Family,Fantasy	A stuffy businessman finds himself trapped ins	Barry Sonnenfeld	Kevin Spacey, Jennifer Garner, Robbie Amell,Ch	2016

- Getting info about your data
 - .info() should be one of the very first commands you run after loading your data

movies_df.info()

```
<class 'pandas.core.frame.DataFrame'>
Index: 1000 entries, Guardians of the Galaxy to Nine Lives
Data columns (total 11 columns):
Rank
                      1000 non-null int64
Genre
                      1000 non-null object
Description
                      1000 non-null object
Director
                      1000 non-null object
                      1000 non-null object
Actors
Year
                      1000 non-null int64
Runtime (Minutes)
                      1000 non-null int64
                      1000 non-null float64
Rating
                      1000 non-null int64
Votes
                      872 non-null float64
Revenue (Millions)
                      936 non-null float64
Metascore
dtypes: float64(3), int64(4), object(4)
memory usage: 93.8+ KB
```

 .info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

 Another fast and useful attribute is .shape, which outputs just a tuple of (rows, columns)

```
movies_df.shape (1000, 11)
```

- Note that .shape has no parentheses and is a simple tuple of format (rows, columns). So we have 1000 rows and 11 columns in our movies DataFrame.
- You'll be going to .shape a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

Handling duplicates

 This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.

```
temp_df = movies_df.append(movies_df)

temp_df.shape

(2000, 11)
```

```
temp_df = temp_df.drop_duplicates()

or
temp_df.drop_duplicates(inplace=True)

temp_df.shape

(1000, 11)
```

- Another important argument for drop_duplicates() is keep, which has three possible options:
 - first: (default) Drop duplicates except for the first occurrence
 - last: Drop duplicates except for the last occurrence
 - False: Drop all duplicates

```
temp_df = movies_df.append(movies_df) # make a new copy
temp_df.drop_duplicates(inplace=True, keep=False)
temp_df.shape (0, 11)
```

Since all rows were duplicates, keep=False dropped them all resulting in zero rows being left over.

Column Cleanup

- Many times datasets will have verbose column names with symbols, upper and lowercase words, spaces, and typos.
- To make selecting data by column name easier we can spend a little time cleaning up their names.
- Print the column names of our dataset.

```
movies_df.columns
```

```
Index(['Rank', 'Genre', 'Description',
'Director', 'Actors', 'Year', 'Runtime
(Minutes)', 'Rating', 'Votes', 'Revenue
(Millions)', 'Metascore'],
dtype='object')
```

Use .rename() method to rename certain or all columns via a dict.

```
Index(['Rank', 'Genre',
'Description', 'Director',
'Actors', 'Year', 'Runtime',
'Rating', 'Votes',
'Revenue_millions',
'Metascore'], dtype='object') 31
```

 What if we want to lowercase all names? Instead of using .rename() we could also set a list of names to the columns like so:

```
movies_df.columns = ['rank', 'genre',
'description', 'director', 'actors',
'year', 'runtime', 'rating', 'votes',
'revenue_millions', 'metascore']
movies_df.columns
```

```
Index(['rank', 'genre',
'description', 'director',
'actors', 'year', 'runtime',
'rating', 'votes',
'revenue_millions',
'metascore'], dtype='object')
```

• But that's too much work. Instead of just renaming each column manually we can do a list comprehension:

```
movies_df.columns = [col.lower() for col
in movies_df]
movies_df.columns
```

```
Index(['rank', 'genre',
'description', 'director',
'actors', 'year', 'runtime',
'rating', 'votes',
'revenue_millions',
'metascore'], dtype='object32)
```

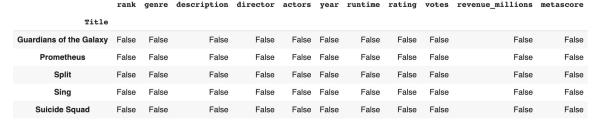
- When exploring data, you'll most likely encounter missing or null values, which are essentially placeholders for non-existent values.
 - Python's None or NumPy's np.nan.
- Two options
 - Get rid of rows or columns with nulls
 - Replace nulls with non-null values, a technique known as imputation

 Let's calculate to total number of nulls in each column of our dataset.

The first step is to check which cells in our DataFrame are

null:

movies_df.isnull()



isnull() returns a DataFrame where each cell is either True or False depending on that cell's null status.

 Then to count the number of nulls in each column we use an aggregate function for summing:

movies_df.isnull().sum()

```
rank 0 genre 0 description 0
director 0 actors 0 year 0
runtime 0 rating 0 votes 0
revenue_millions 128 metascore
64 dtype: int64
```

Remove null values

- Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of your data and its context.
- Overall, removing null data is only suggested if you have a small amount of missing data.

movies_df.dropna()

This operation will delete any **row** with at least a single null value, but it will return a new DataFrame without altering the original one. You could specify **inplace=True** in this method as well.

movies_df.dropna(axis=1)

Drop columns with null values

Check the reference by yourself for imputation method

- Note that before you apply .isnull() and .dropna() methods, you need to make sure that missing values are represented properly (either None or np.nan).
- We can convert special missing values into such standard forms by specifying the na_values attribute when calling pd.read_csv().
 - na_values: scalar, str, list-like, or dict, optional
 - Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: ", '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.
 - For instance: "?", "\whitespace", "&", and ":".

Check Data Statistics

- Understanding your variables
 - Using describe() on an entire DataFrame we can get a summary of the distribution of continuous variables.

movies_df.describe()



 By using the correlation method .corr() we can generate the relationship between each continuous variable:

movies_df.corr()

	rank	year	runtime	rating	votes	revenue_millions
rank	1.000000	-0.261605	-0.221739	-0.219555	-0.283876	-0.252996
year	-0.261605	1.000000	-0.164900	-0.211219	-0.411904	-0.117562
runtime	-0.221739	-0.164900	1.000000	0.392214	0.407062	0.247834
rating	-0.219555	-0.211219	0.392214	1.000000	0.511537	0.189527
votes	-0.283876	-0.411904	0.407062	0.511537	1.000000	0.607941
revenue_millions	-0.252996	-0.117562	0.247834	0.189527	0.607941	1.000000
metascore	-0.191869	-0.079305	0.211978	0.631897	0.325684	0.133328

Check Data Statistics

Understanding your variables

 describe() can also be used on a categorical variable to get the count of rows, unique count of categories, top category,

and freq of top category:

movies_df['genre'].describe()

```
count 1000
unique 207
top Action, Adventure, Sci-Fi
freq 50
Name: genre, dtype: object
```

- .value_counts() can tell us the frequency of all values

in a column:

movies_df['genre'].value_counts().head(10)

Action, Adventure, Sci-Fi	50
Drama	48
Comedy, Drama, Romance	35
Comedy	32
Drama, Romance	31
Action,Adventure,Fantasy	27
Comedy, Drama	27
Animation, Adventure, Comedy	27
Comedy, Romance	26
Crime,Drama,Thriller	24
Name: genre, dtype: int64	

 It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.

By columns

```
genre_col = movies_df['genre']
type(genre_col)
```

```
genre_col = movies_df[['genre']]
type(genre_col)
```

```
subset = movies_df[['genre', 'rating']]
subset.head()
```

```
pandas.core.series.Series
```

pandas.core.frame.DataFrame

	genre	rating
Title		
Guardians of the Galaxy	Action,Adventure,Sci-Fi	8.1
Prometheus	Adventure, Mystery, Sci-Fi	7.0
Split	Horror,Thriller	7.3
Sing	Animation,Comedy,Family	7.2
Suicide Squad	Action,Adventure,Fantasy	6.2

- By rows: two options
 - .loc **loc**ates by name
 - iloc- locates by numerical index

Single Row

```
prom = movies_df.loc["Prometheus"]
```

1 is the numerical index of Prometheus:

```
prom = movies_df.iloc[1]
```

Multiple Rows

```
movie_subset = movies_df.loc['Prometheus':'Sing']
```

movie_subset = movies_df.iloc[1:4]

One important distinction between using .loc and .iloc to select multiple rows is that .loc includes the movie *Sing* in the result, but when using .iloc we're getting rows 1:4 but the movie at index 4 (*Suicide Squad*) is not included -- similar to Python list slicing

Conditional selections

'Ridley Scott')]

— For example, what if we want to filter our movies DataFrame to show only films directed by Ridley Scott or films with a rating greater than or equal to 8.0?

```
condition = (movies_df['director'] == "Ridley Scott")
condition.head()
```

```
Title
Guardians of the Galaxy False
Prometheus True
Split False
Sing False
Suicide Squad False
Name: director, dtype: bool
```

Similar to isnull(), this returns a Series of True and False values: True for films directed by Ridley Scott and False for ones not directed by him.

```
movies_df[movies_df['director'] == "Ridley Scott"]

movies_df[movies_df['rating'] >= 8.6]

movies_df[(movies_df['director'] == 'Christopher Nolan') | (movies_df['director'] == 'Christo
```

Say we want all movies that were released between 2005 and 2010, have a rating above 8.0, but made below the 25th percentile in revenue.

```
movies_df[
  ((movies_df['year'] >= 2005) & (movies_df['year'] <= 2010))
  & (movies_df['rating'] > 8.0)
  & (movies_df['revenue_millions'] < movies_df['revenue_millions'].quantile(0.25))
]</pre>
```

Apply Functions

- For example, we want to convert movies with an 8.0 or greater to a string value of "good" and the rest to "bad" and use this transformed values to create a new column.
- It is possible to iterate over a DataFrame or Series as you would with a list, but doing so — especially on large datasets — is very slow.
- Using *apply()* will be much faster than iterating manually over rows because pandas is utilizing vectorization.
 - Vectorization: a style of computer programming where operations are applied to whole arrays instead of individual elements Wikipedia

Apply Functions

```
def rating_function(x):
    if x >= 8.0:
        return "good"
    else:
        return "bad"
```

```
movies_df["rating_category"] = movies_df["rating"].apply(rating_function)
```

The .apply() method passes every value in the rating column through the rating_function and then returns a new Series. This Series is then assigned to a new column called rating category.

You can also use anonymous functions as well. This lambda function achieves the same result as rating_function:

```
movies_df["rating_category"] = movies_df["rating"].apply(lambda x: 'good' if x >= 8.0 else 'bad')
```

Reading References

Chapter 3 of [T1]